



Introduction to Digital Peripherals

Dheeraj Kamath - Applications Engineer

WW 20 17



Objective

- By the end of this training, you will
 - ✓ Learn the digital peripherals available in Cypress' PSoC 6
 - ✓ Understand their basic functions
 - ✓ Learn how to use them in an application using ModusToolbox

Hardware:

PSoC6 BLE Pioneer Kit

PSoC6 WIFI-BT Pioneer Kit

CY8CPROTO-062-4343W PSoC6 Prototyping Kit

Software:

ModusToolbox 2.1

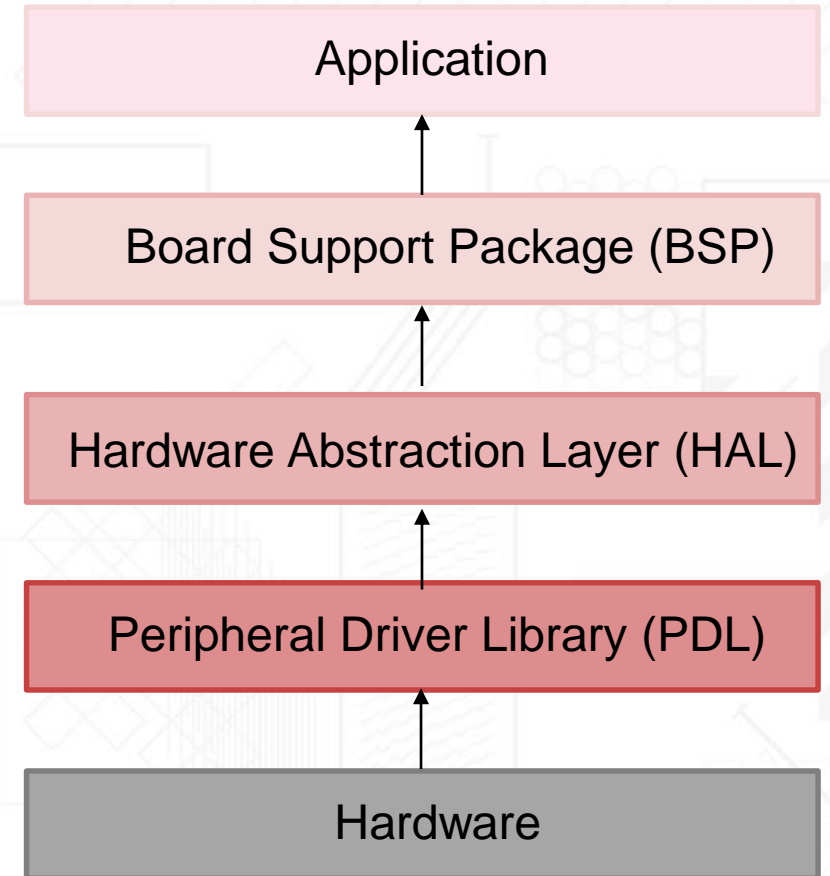
Agenda

- Recap
- Quick overview: ModusToolbox 2.1
- Introduction
 - Digital Architecture
- Smart-IO
- TCPWM
- SCBs
- Exercises

Recap

In the previous training we understood:

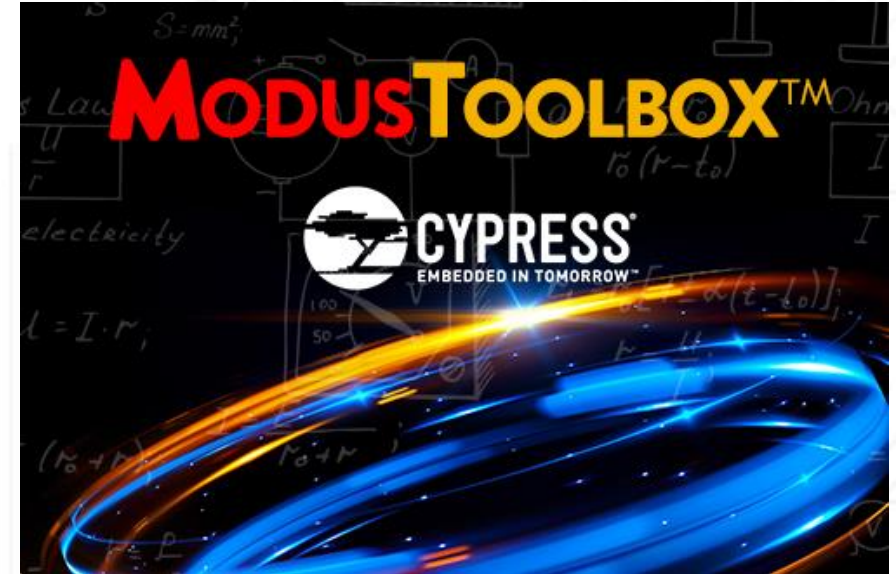
- ✓ what ModusToolbox is
- ✓ what it comprises
- ✓ How to create a project
- ✓ the directory structure
- ✓ different tools and configurators
- ✓ terminologies like HAL, BSP, PDL
- ✓ different ecosystems supported by ModusToolbox



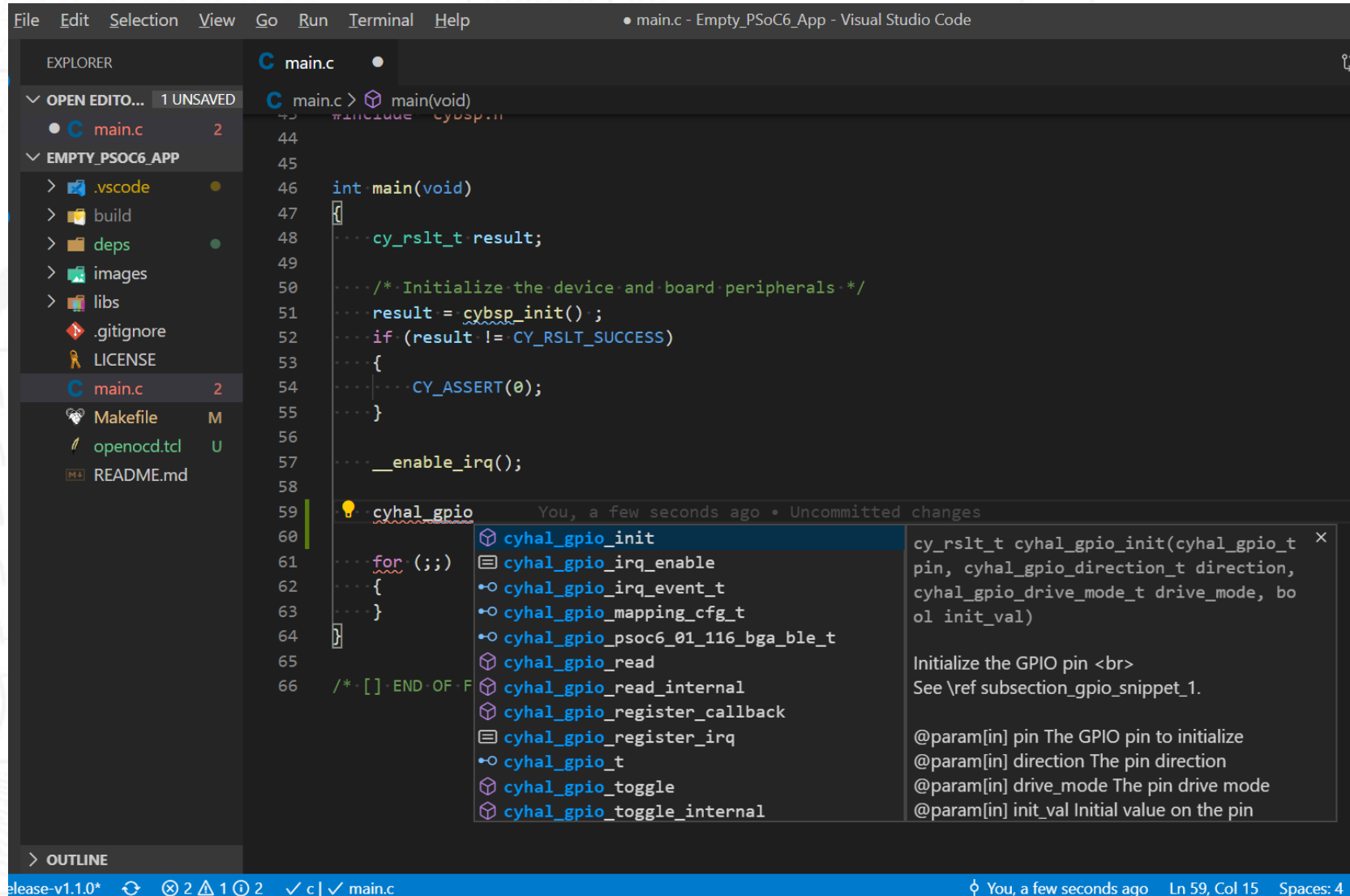
ModusToolbox 2.1

What's new?

- No longer called ModusToolbox IDE
- Proxy Handling Improvements
- Upgrades to Tools and Configurators
- OpenOCD 3.0
- Offline Support Package
- Support for third party IDEs
 - ✓ IAR Embedded Workbench
 - ✓ Keil ARM-MDK
 - ✓ Visual Studio Code (VSCode)



Visual Studio Code



The screenshot shows the Visual Studio Code interface in dark mode. The Explorer sidebar on the left displays the project structure for 'EMPTY_PSOC6_APP', including files like .vscode, build, deps, images, libs, .gitignore, LICENSE, main.c (2 lines), Makefile (M), openocd.tcl (U), and README.md. The main editor window shows the code for 'main.c', which includes a Cython header and a main function that initializes the device and board peripherals, checks the result of cybsp_init(), asserts success, and enables interrupts. A popup window is visible over the code, showing the signature for 'cyhal_gpio_init' and a list of related functions: cyhal_gpio_irq_enable, cyhal_gpio_irq_event_t, cyhal_gpio_mapping_cfg_t, cyhal_gpio_psoc6_01_116_bga_ble_t, cyhal_gpio_read, cyhal_gpio_read_internal, cyhal_gpio_register_callback, cyhal_gpio_register_irq, cyhal_gpio_t, cyhal_gpio_toggle, and cyhal_gpio_toggle_internal. The status bar at the bottom indicates the current file is 'main.c' at line 59, column 15, with 4 spaces.

```
File Edit Selection View Go Run Terminal Help
• main.c - Empty_PSOC6_App - Visual Studio Code

EXPLORER
OPEN EDITOR... 1 UNSAVED
• main.c 2
EMPTY_PSOC6_APP
  .vscode
  build
  deps
  images
  libs
  .gitignore
  LICENSE
  main.c 2
  Makefile M
  openocd.tcl U
  README.md

main.c
43 #include <cybsp.h>
44
45
46 int main(void)
47 {
48     cy_rslt_t result;
49
50     /* Initialize the device and board peripherals */
51     result = cybsp_init();
52     if (result != CY_RSLT_SUCCESS)
53     {
54         CY_ASSERT(0);
55     }
56
57     __enable_irq();
58
59     cyhal_gpio
60     cyhal_gpio_init
61     for (;;)
62     {
63     }
64
65     cyhal_gpio_read
66     /* [] END OF FILE */

You, a few seconds ago • Uncommitted changes
cyhal_gpio_init
cyhal_gpio_irq_enable
cyhal_gpio_irq_event_t
cyhal_gpio_mapping_cfg_t
cyhal_gpio_psoc6_01_116_bga_ble_t
cyhal_gpio_read
cyhal_gpio_read_internal
cyhal_gpio_register_callback
cyhal_gpio_register_irq
cyhal_gpio_t
cyhal_gpio_toggle
cyhal_gpio_toggle_internal

cy_rslt_t cyhal_gpio_init(cyhal_gpio_t
pin, cyhal_gpio_direction_t direction,
cyhal_gpio_drive_mode_t drive_mode, bo
ol init_val)

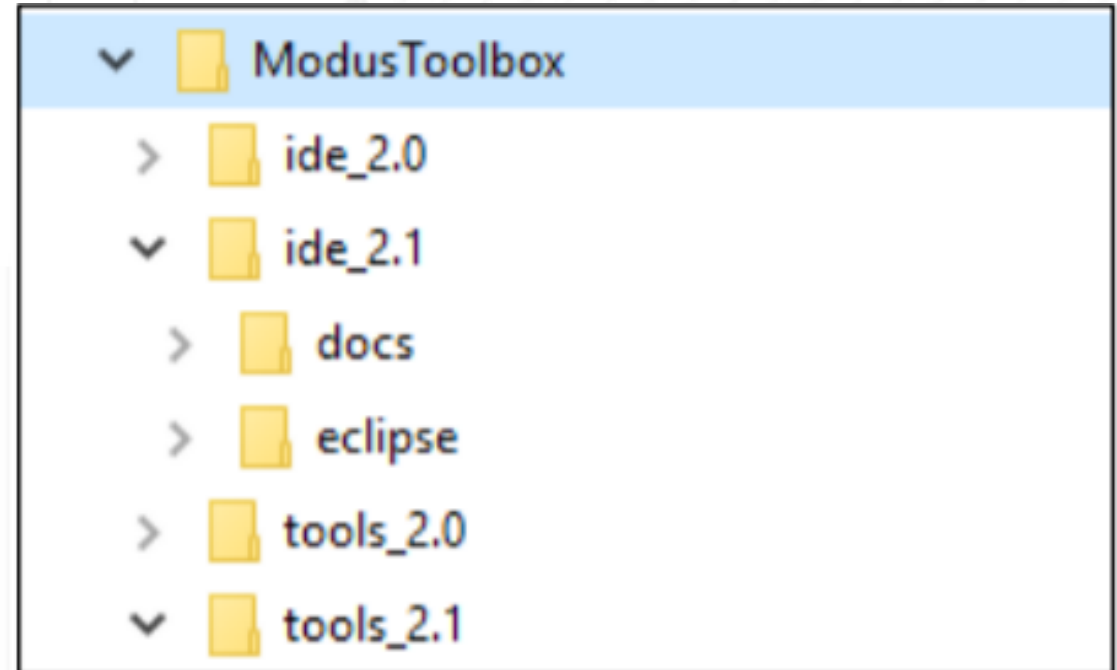
Initialize the GPIO pin <br>
See \ref subsection_gpio_snippet_1.

@param[in] pin The GPIO pin to initialize
@param[in] direction The pin direction
@param[in] drive_mode The pin drive mode
@param[in] init_val Initial value on the pin
```

Yes, now you
can work in
dark mode!!!

Product Versioning

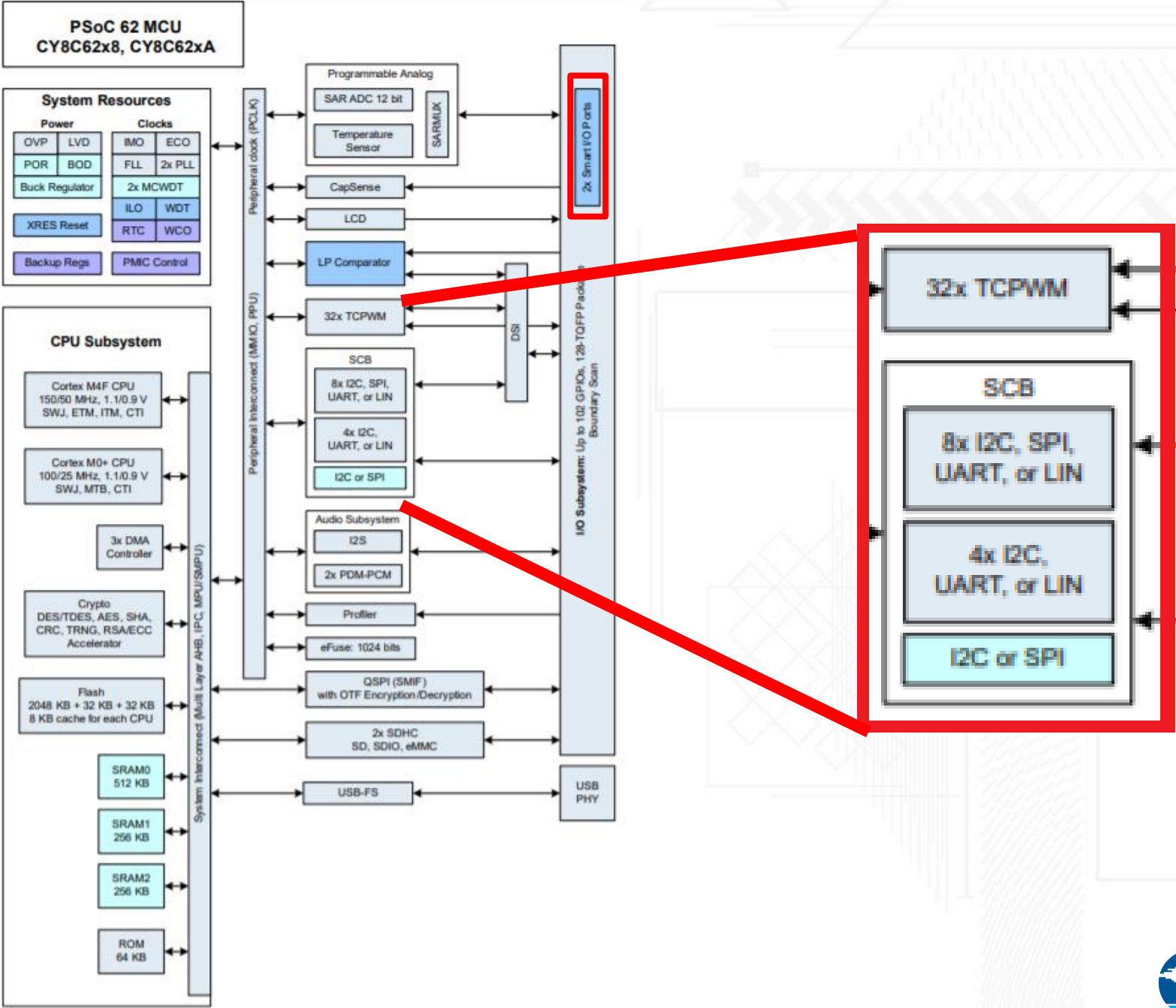
- The ModusToolbox installation package is versioned as MAJOR.MINOR.PATCH. The file located at /ModusToolbox/tools_2.1/version-2.1.0.xml also indicates the build number
- Multiple versions installed in parallel in the same ModusToolbox directory
- Flexibility to use the version you want
 - ✓ Using Application Makefile
 - ✓ Set CY_TOOLS_PATH in environment variable



Introduction



PSoC6 Device Architecture



PSoC6 Digital Peripherals

- Programmable Digital
 - ✓ Smart-IO - programmable logic fabric that enables Boolean operations on signals passing through it
- Fixed-Function Digital
 - ✓ Timer/Counter/PWM Block (TCPWM) –
 - Timer-counter with compare
 - Timer-counter with capture
 - Quadrature decoding
 - Pulse width modulation (PWM)
 - ✓ Serial Communication Blocks (SCB) – digital block that is configurable as UART, I2C or SPI interfaces
 - ✓ USB, QSPI, SD Host Controller (out of scope for this training)

2 x Smart-IO Ports

32 x TCPWM

SCB

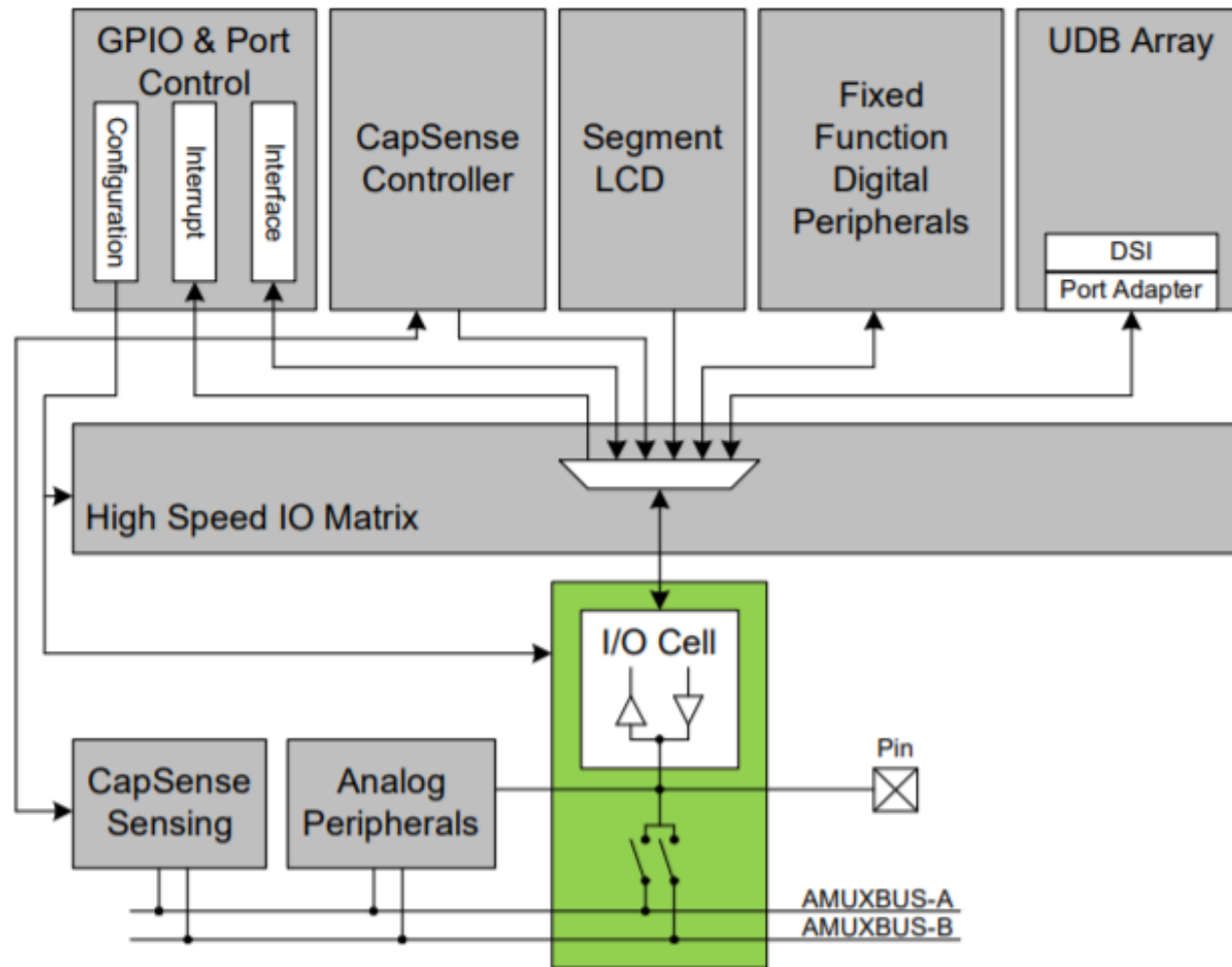
8x I2C, SPI,
UART, or LIN

4x I2C,
UART, or LIN

I2C or SPI

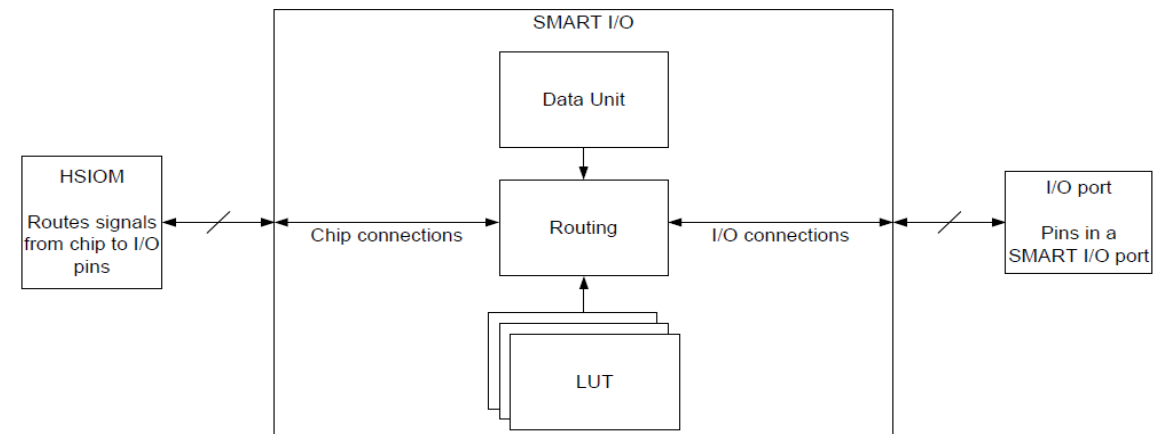
High Speed Input Output Matrix (HSIOM)

- Contains multiplexers to connect between the selected peripheral and the pin.



Smart-IO

- The Smart I/O block sits between the GPIO pins and the high-speed I/O matrix (HSIOM) and is dedicated to a single port.
- Smart I/O supports:
 - ✓ Deep Sleep operation
 - ✓ Boolean operations without CPU intervention
 - ✓ Asynchronous or synchronous (clocked) operation
- Three selectable input sources
 - ✓ another LUT
 - ✓ an internal resource
 - ✓ an external signal from a GPIO pin



Smart-IO Configurator

C:/Users/ddka/CDC_Training/mtb_02_ex01_smartio_rgb/libs/TARGET_CY8CKIT-062-BLE/COMPONENT_BSP_DESIGN_MODUS/design.modus - Smart I/O Configurator 2.1

File View Help

Routing LUT 0 LUT 1 LUT 2 LUT 4

Port: Port 9 (Smart I/O 9) Clock: Peripheral clock divider (Active) Clock divider: 24.5 bit Divider 0 clk [USED] Show Routing Matrix Clear

Chip 7 Bypass
Chip 6 Bypass
Chip 5 Bypass
Chip 4 None
Chip 3 Bypass
Chip 2 None
Chip 1 Input(Async)
TCPWM[1] 16-bit Counter 20 pwm_n (PWM) [USED]
Chip 0 None

I/O 7 Bypass
I/O 6 Bypass
I/O 5 Bypass
I/O 4 Output
I/O 3 Bypass
I/O 2 Output
I/O 1 Output
I/O 0 Output

0 1 2 Data Unit
TR 0: rst Constant 0
TR 1: en Constant 0
TR 2: [UNUSED] Constant 0

LUT 0 LUT 1 LUT 2 LUT 3 LUT 4 LUT 5 LUT 6 LUT 7

Chip 1
Chip 1
Chip 1
None
LUT1
LUT2
LUT4
None
None
None
None
None
None
None
None

Ready

The image shows the Smart-IO Configurator 2.1 software interface. At the top, the file path is displayed: C:/Users/ddka/CDC_Training/mtb_02_ex01_smartio_rgb/libs/TARGET_CY8CKIT-062-BLE/COMPONENT_BSP_DESIGN_MODUS/design.modus. The interface includes a menu bar (File, View, Help) and a toolbar with options for Routing, LUT 0, LUT 1, LUT 2, and LUT 4. A status bar at the top indicates the current port (Port 9 (Smart I/O 9)), clock (Peripheral clock divider (Active)), and clock divider (24.5 bit Divider 0 clk [USED]). There are buttons for 'Show Routing Matrix' and 'Clear'. The main area is a routing matrix with rows for chips (Chip 7 to Chip 0) and columns for I/Os (I/O 7 to I/O 0). Green lines and dots indicate the routing connections. On the left, a list of chips is shown with their configurations: Chip 7 (Bypass), Chip 6 (Bypass), Chip 5 (Bypass), Chip 4 (None), Chip 3 (Bypass), Chip 2 (None), Chip 1 (Input(Async) with TCPWM[1] 16-bit Counter 20 pwm_n (PWM) [USED]), and Chip 0 (None). On the right, a list of I/Os is shown with their configurations: I/O 7 (Bypass), I/O 6 (Bypass), I/O 5 (Bypass), I/O 4 (Output), I/O 3 (Bypass), I/O 2 (Output), I/O 1 (Output), and I/O 0 (Output). At the bottom, there are sections for 'Data Unit' (TR 0: rst, TR 1: en, TR 2: [UNUSED]) and a list of LUTs (LUT 0 to LUT 7) with their configurations: LUT 0 (Chip 1), LUT 1 (Chip 1), LUT 2 (Chip 1), LUT 3 (None), LUT 4 (LUT1, LUT2, LUT4), LUT 5 (None), LUT 6 (None), and LUT 7 (None). The status bar at the bottom left shows 'Ready'.

Lookup Table (LUT)

C:/Users/ddka/CDC_Training/mtb_02_ex01_smartio_rgb/libs/TARGET_CY8CKIT-062-BLE/COMPONENT_BSP_DESIGN_MODUS/design.mod... — □ ×

File View Help

Routing LUT 0 LUT 1 LUT 2 LUT 4

Mode: Sequential (gated) output ▼

TR0 TR1 TR2

Lookup Table

Out

clk

Mapping:

TR 2: (LUT4)	TR 1: (LUT2)	TR 0: (LUT1)	Set Bit
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Output: Hex ▼ 0x55

Ready

Using Smart-IO in your Application

Steps:

- Go to design.modus file and enable Smart-IO. Use the Smart-IO configurator to define the inputs, the outputs and the logical operations to be performed.
- Make use of these APIs (basic) to start the Smart-IO block

`Cy_SmartIO_Init(SMARTIO_HW, &SMARTIO_config);` - Initializes the SMART-IO Block

`Cy_SmartIO_Enable(SMARTIO_HW);` - Enables it

Refer to the code example "*Ramping LED using Smart-IO*" for more information.

Note: No HAL yet, support only through PDL.

Smart-IO Configurator

Exercise 1:

Configure a PWM to generate a frequency of 1Hz with 50% duty cycle. Route this signal to pin 9[0] using the Smart-IO Block. In firmware read the output on pin 9[0] and write to LED9 (P13_7) and observe the LED blinking every second.

Exercise 2:

Create a LUT such that it functions as a 8-bit counter. Route the three output signals of the LUT to the RGB LEDs and observe the colors as shown in the below table.

LUT3	LUT2	LUT1	Color
0	0	0	OFF
0	0	1	RED
0	1	0	GREEN
0	1	1	YELLOW
1	0	0	BLUE
1	0	1	PINK
1	1	0	INDIGO
1	1	1	WHITE

Refer [mtb_02_ex01_smartio_rgb](#) project which implements the solution to both the exercises.

Smart-IO Configurator

Exercise 2 LUT Logic Explained:

Present State			Next State		
LUT4	LUT2	LUT1	LUT4	LUT2	LUT1
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

Refer `mtb_02_ex01_smartio_rgb` project which implements the solution to both the exercises.

Timer Counter Pulse Width Modulation

- Multi-functional, configurable digital block containing 32 counters. Each can be 16 or 32-bit wide.
- Modes:
 - ✓ Counter – counts events, for e.g., number of pulse edges
 - ✓ Timer – sets up a counter to generate time intervals
 - ✓ PWM – Generates pulses based on the duty cycle, period and compare values
- Up, Down, and Up/Down counting modes
- Clock prescaling (division by 1, 2, 4, ... 64, 128)

TCPWM[0] 32-bit Counter 1 (RED_PWM) - Parameters - Device Configurator 2.1

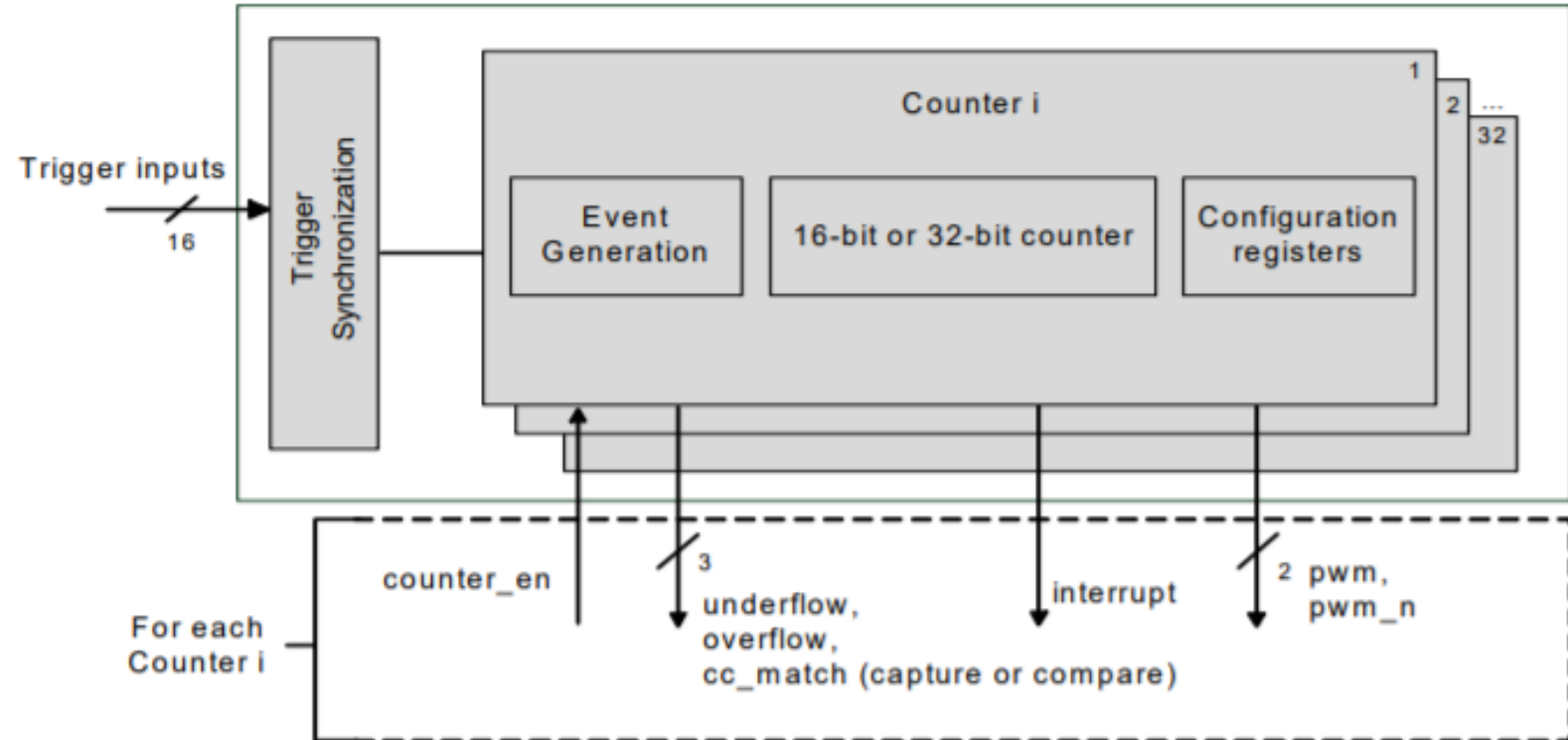
Enter filter text...

Name	Value
Peripheral Documentation	
? Configuration Help	Open PWM (TCPWM) Documentation
General	
? PWM Mode	PWM
? Clock Prescaler	Divide by 1
? PWM Resolution	32-bits
? PWM Alignment	Left Aligned
? Run Mode	Continuous
Period	
? Enable Period Swap	<input type="checkbox"/>
? Period	9999
Compare	
? Enable Compare Swap	<input type="checkbox"/>
? Compare	5000
Interrupts	
? Interrupt Source	None
Inputs	
? Clock Signal	16 bit Divider 0 clk [USED]
? Count Input	Disabled
? Kill Input	Disabled
? Reload Input	Disabled
? Start Input	Disabled
? Swap Input	Disabled
PWM Output Polarity	
? Invert PWM Output	<input type="checkbox"/>
? Invert PWM_n Output	<input type="checkbox"/>
Outputs	
? PWM (line)	<unassigned> ...
? PWM_n (line_compl)	P0[3] digital_out [USED] ...
? Overflow	<unassigned> ...
? Underflow	<unassigned> ...
? Compare (cc_match)	<unassigned> ...
Advanced	
? Store Config in Flash	<input checked="" type="checkbox"/>

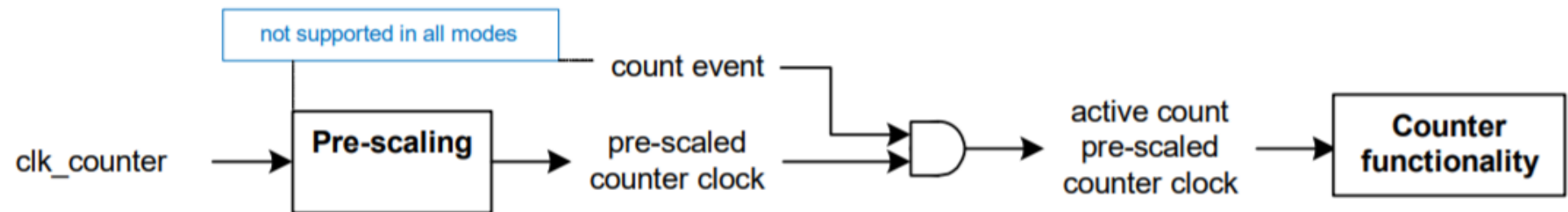
Timer Counter Pulse Width Modulation

Provides two interfaces:

- I/O signal interface:
 - ❖ Consists of input triggers:
 - ✓ Reload
 - ✓ Start
 - ✓ Stop
 - ✓ Count
 - ✓ Capture
 - ❖ Output signals:
 - ✓ pwm
 - ✓ pwm_n
 - ✓ overflow (OV)
 - ✓ underflow (UN)
 - ✓ capture/compare (CC)).
- Interrupts: Provides interrupt request signals from each counter, based on TC or CC conditions



Counter Functionality



Configurable Modes

Mode	MODE Field [26:24]	Description
Timer	000	The counter increments or decrements by '1' at every <code>clk_counter</code> cycle in which a count event is detected. The Compare/Capture register is used to compare the count.
Capture	010	The counter increments or decrements by '1' at every <code>clk_counter</code> cycle in which a count event is detected. A capture event copies the counter value into the capture register.
Quadrature	011	Quadrature decoding. The counter is decremented or incremented based on two phase inputs according to an X1, X2, or X4 decoding scheme.
PWM	100	Pulse width modulation.

Understanding HAL

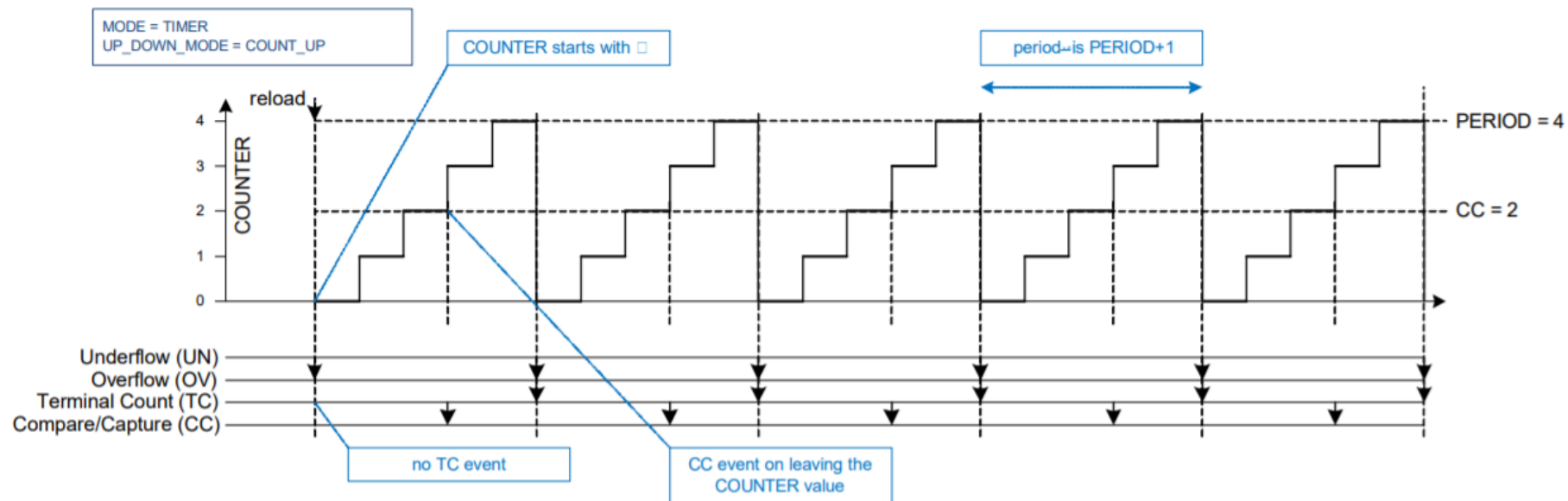
- It is a generic interface that can be used across multiple product families
- The focus is on ease-of-use and portability

API Structure

- `_init` function – Allocates a block, configures it and enables it.
- `_free` function – Disables a block, releases resources
- Other functions – provide block specific functionality

Timer

- Increments/decrements a counter between 0 and the value stored in the PERIOD register.
- Used for:
 - ✓ Timing a specific delay
 - ✓ Counting the occurrence of a specific event



Using Timer in your Application

Steps:

Using HAL APIs:

First configure the timer parameters using `cyhal_timer_cfg_t` and then initialize and enable it as shown:

```
const cyhal_timer_cfg_t led_blink_timer_cfg =  
{  
    .compare_value = 0,                /* Timer compare value, not used */  
    .period = LED_BLINK_TIMER_PERIOD, /* Defines the timer period */  
    .direction = CYHAL_TIMER_DIR_UP,  /* Timer counts up */  
    .is_compare = false,              /* Don't use compare mode */  
    .is_continuous = true,            /* Run timer indefinitely */  
    .value = 0                        /* Initial value of counter */  
};  
  
cyhal_timer_init(&led_blink_timer, NC, NULL);  
  
cyhal_timer_configure(&led_blink_timer, &led_blink_timer_cfg);  
  
cyhal_timer_set_frequency(&led_blink_timer, LED_BLINK_TIMER_CLOCK_HZ);
```

Using Timer in your Application

Additionally you can register the callback functions to be triggered when a specific event occurs using the following APIs:

```
/* Assign the ISR to execute on timer interrupt */  
cyhal_timer_register_callback(&led_blink_timer, isr_timer, NULL);  
  
/* Set the event on which timer interrupt occurs and enable it */  
cyhal_timer_enable_event(&led_blink_timer, CYHAL_TIMER_IRQ_TERMINAL_COUNT,  
7, true);
```

Then you can go ahead and start the timer!

```
/* Start the timer with the configured settings */  
cyhal_timer_start(&led_blink_timer);
```


Using Timer in your Application

Steps:

Using PDL APIs:

First configure the timer parameters using the design.modus file and then use the following APIs to start the timer:

```
/* Initialize the interrupt */  
  
Cy_SysInt_Init(&timer_isr_config, timer_isr);  
NVIC_EnableIRQ(timer_isr_config.intrSrc);  
  
/* Start the timer */  
  
Cy_TCPWM_Counter_Init(TIMER_HW, TIMER_NUM, &TIMER_config);  
Cy_TCPWM_Counter_Enable(TIMER_HW, TIMER_NUM);  
Cy_TCPWM_TriggerStart(TIMER_HW, TIMER_MASK);
```

Counter

- The capture functionality increments and decrements a counter between 0 and PERIOD. When the capture event is activated the counter value COUNTER is copied to CC.
- Used for:
 - ✓ Measuring the width of a pulse
 - ✓ Measuring the frequency of a signal

Using Counter in your Application

Steps:

Using PDL APIs (HAL not supported):

First configure the timer parameters using the design.modus file and then use the following APIs to start the timer:

```
/* Start the timer */  
  
Cy_TCPWM_Counter_Init(COUNTER_HW, COUNTER_NUM, &COUNTER_config);  
  
Cy_TCPWM_Counter_Enable(COUNTER_HW, COUNTER_NUM);  
  
Cy_TCPWM_TriggerStart(COUNTER_HW, COUNTER_MASK);
```

To read the captured values use:

```
Cy_TCPWM_Counter_GetCapture()
```

Timer/Counter

Exercise 3:

Configure a timer to generate an interrupt every 1s and toggle the LED. Use only HAL APIs.

Refer [mtb_02_ex04_timer_hal](#) project which implements the solution to this exercise.

Exercise 4:

Configure a timer to generate an interrupt every 1s and toggle the LED. Use only PDL APIs.

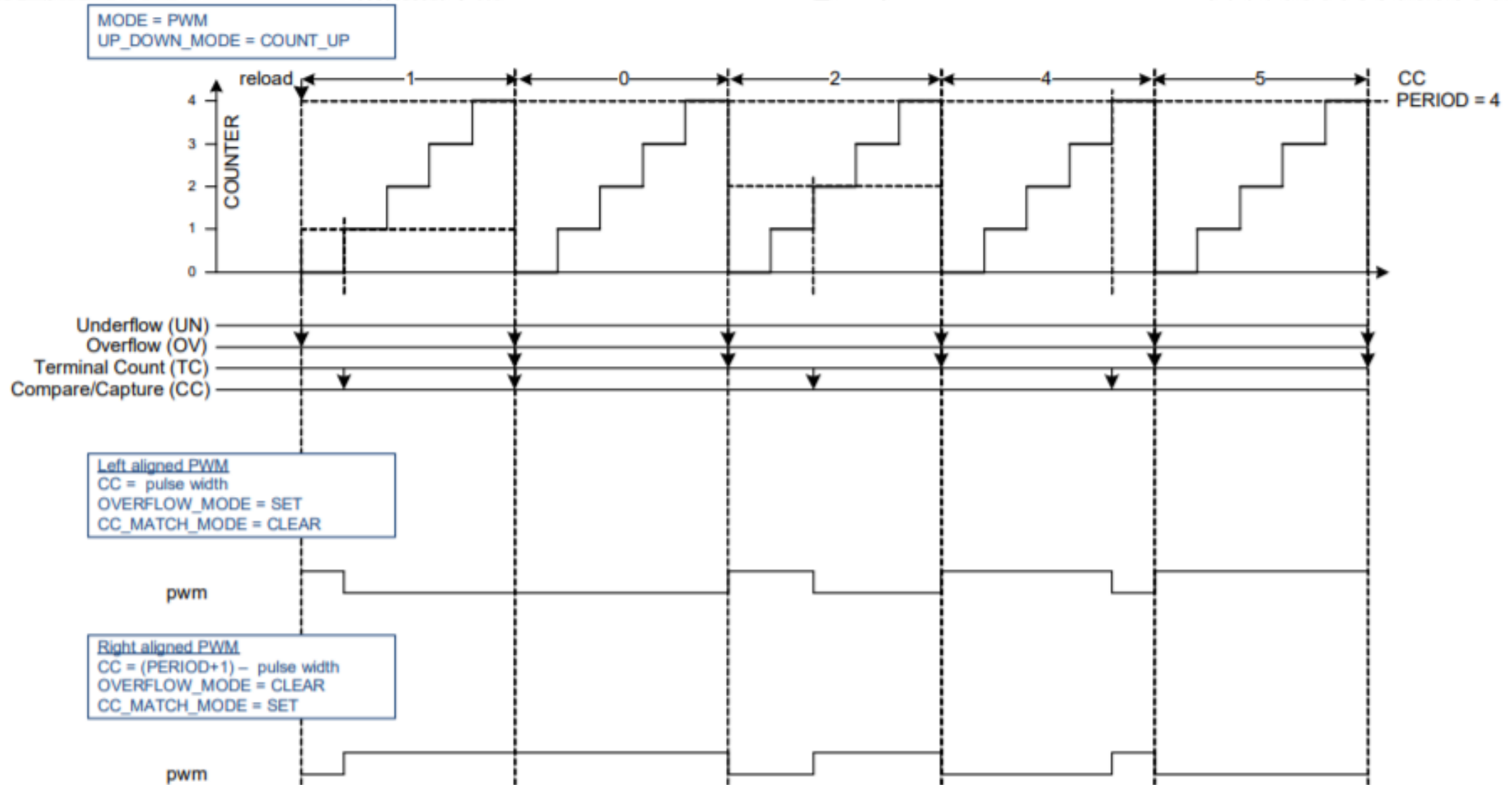
Refer [mtb_02_ex05_timer_pdl](#) project which implements the solution to this exercise. Compare this with the previous exercise. Which do you think was easier?

Exercise 5:

- Generate a 250 Hz signal with any duty cycle using PWM
- Use a Counter to measure counts between PWM pulses
- Print measured frequency to serial terminal using UART

Refer [mtb_02_ex06_counter_dutycycle](#) project which implements the solution to this exercise.

PWM



Using PWM in your Application

Steps:

Using HAL APIs:

Add the following code directly in main.c to interact with the PWM block

```
cyhal_pwm_init(&pwm_obj, CYBSP_USER_LED, NULL);  
cyhal_pwm_set_duty_cycle(&pwm_obj, 50, 1);  
cyhal_pwm_start(&pwm_obj);
```

Using PDL APIs:

Configure a PWM in design.modus file with the required parameters and then call the following APIs:

```
Cy_TCPWM_PWM_Init(PWM_HW, PWM_NUM, &PWM_config);  
Cy_TCPWM_PWM_Enable(PWM_HW, PWM_NUM);  
Cy_TCPWM_TriggerStart(PWM_HW, PWM_MASK);
```

PWM

Exercise 6:

Configure a PWM to generate a frequency of 1Hz with 50% duty cycle using HAL and PDL APIs. Check if there is a conflict.

Solution key: Refer `mtb_02_ex02_pwm_blinkyled` project for this solution.

Exercise 7:

Configure a PWM to increase the brightness of the LED to its maximum and then decrement to its lowest. Hint: Vary the duty cycle every 500ms to observe the output on the LED.

Solution key: Refer `mtb_02_ex03_pwm_brightness_control` project for this solution.

Serial Communication Block

- Multi-functional, configurable digital communication block
- Can be made to function as communication components:
 - ✓ I2C
 - ✓ SPI
 - ✓ UART
- Standard SPI master and slave functionality with Motorola, Texas Instruments, and National Semiconductor protocols Standard
- UART functionality
- Standard I2C master and slave functionality
- Trigger outputs for connection to DMA
- Multiple interrupt sources to indicate status of FIFOs and transfers

Serial Communication Block (SCB) 5 - Parameters - Device Configurator 2.1

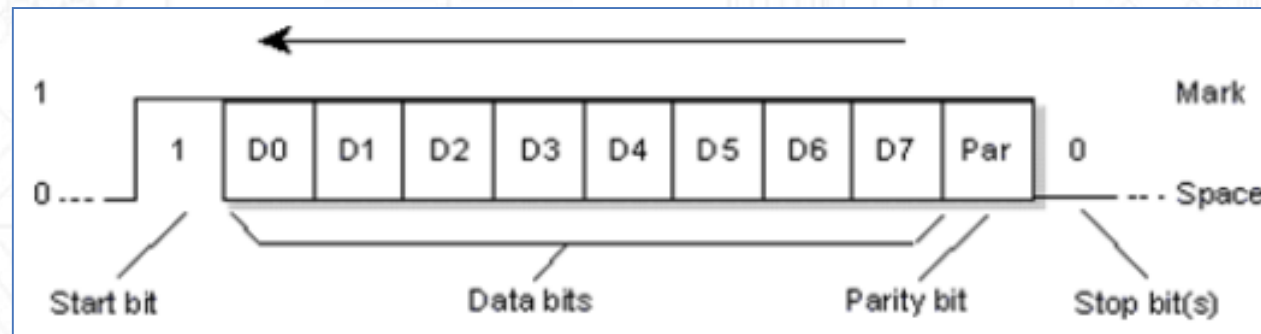
Enter filter text...

Name	Value
Configuration Help	Open UART (SCB) Documentation
General	
Com Mode	Standard
Baud Rate (bps)	115200
Oversample	8
Bit Order	LSB First
Data Width	8 bits
Parity	None
Stop Bits	1 bit
Enable Digital Filter	<input type="checkbox"/>
Support RS-485	
TX-Enable	<input type="checkbox"/>
Flow Control	
Enable Flow Control	<input type="checkbox"/>
CTS Polarity	Active Low
RTS Polarity	Active Low
RTS Activation Level	63
Connections	
Clock	<unassigned>
RX	<unassigned>
TX	<unassigned>
RX Trigger Output	<unassigned> ...
TX Trigger Output	<unassigned> ...
Actual Baud Rate	
Actual Baud Rate (bps)	<input type="text"/>
Baud Rate Accuracy (%)	<input type="text"/>
Clock Frequency	<input type="text"/>
Trigger Level	
RX FIFO Level	63
TX FIFO Level	63
Multi Processor Mode	
Enable Multi Processor Mode	<input type="checkbox"/>
Address	0

Serial Communication Block

UART

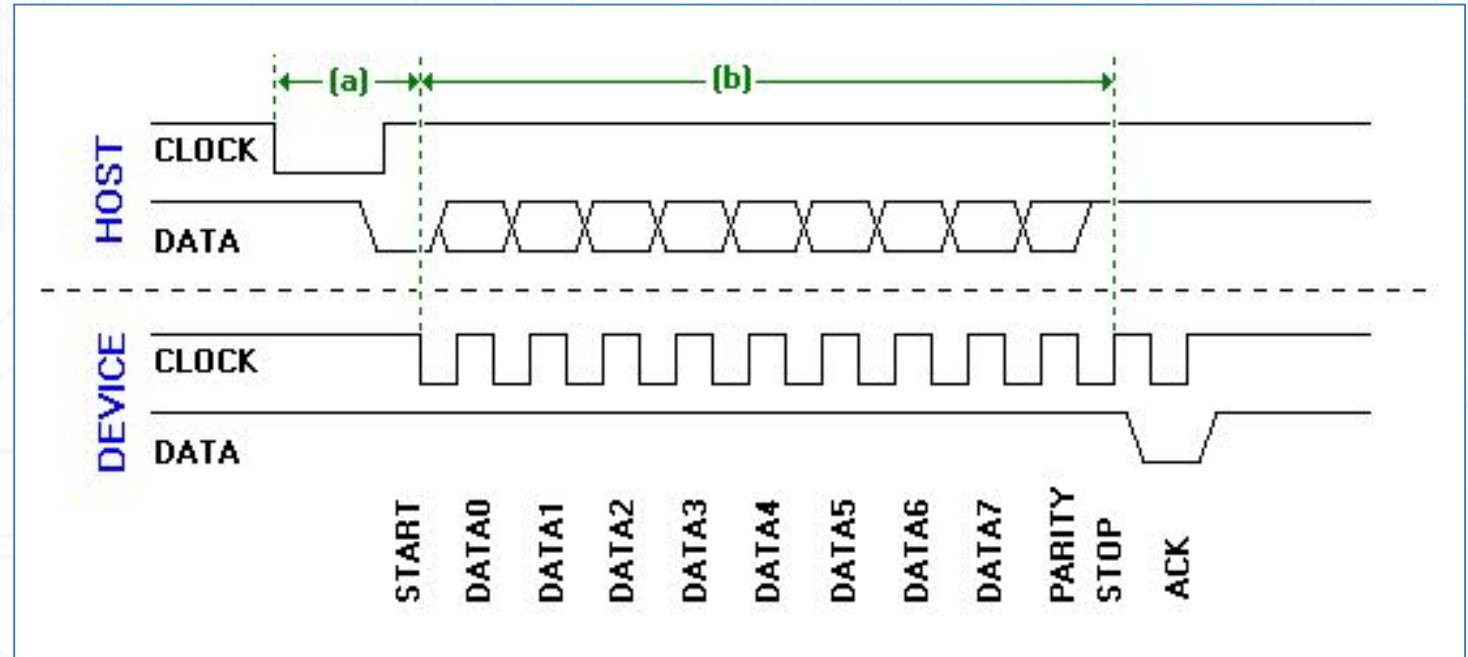
- Universal asynchronous transmitter and receiver
- Half duplex, full duplex,, only TX and only RX modes
- Two wire – Transmit (TX) and Receive (RX)
- No Clock line
- Typically used baud rates – 9600 to 115200 bps
- Additional pins – flow control functionality
- Usually between two devices



Serial Communication Block

UART

- Universal asynchronous transmitter and receiver
- Half duplex, full duplex,, only TX and only RX modes
- Two wire – Transmit (TX) and Receive (RX)
- No Clock line
- Typically used baud rates – 9600 to 115200 bps
- Additional pins – flow control functionality
- Usually between two devices



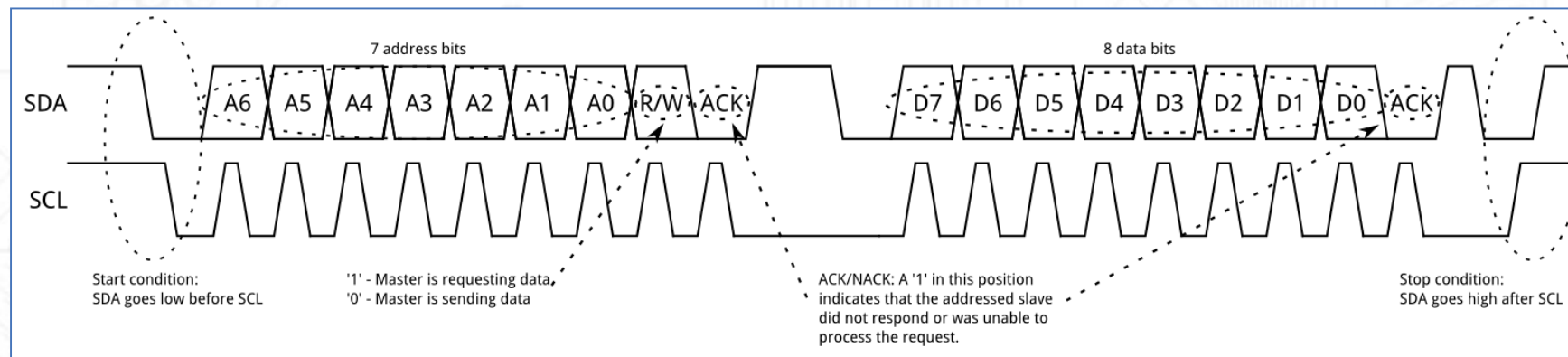
Serial Communication Block

UART

- Universal asynchronous transmitter and receiver
- Half duplex, full duplex,, only TX and only RX modes
- Two wire – Transmit (TX) and Receive (RX)
- No Clock line
- Typically used baud rates – 9600 to 115200 bps
- Additional pins – flow control functionality
- Usually between two devices

I2C

- Inter-integrated circuit (IIC / I²C)
- Half duplex protocol
- Two wire – Serial Data (SDA) and Serial Clock (SCL)
- Typically used clock rates – 100 kHz to 400 kHz
- A master can talk to 127 slaves



Serial Communication Block

UART

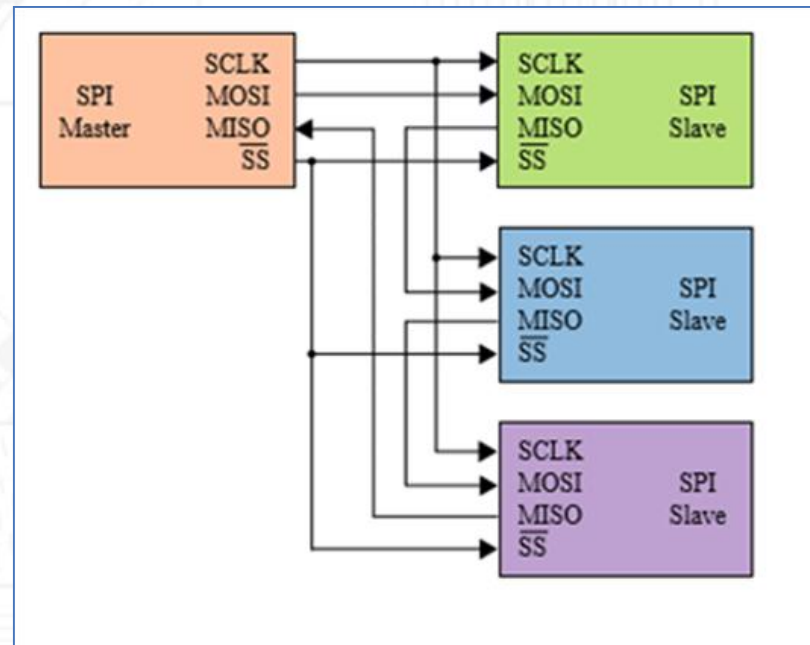
- Universal asynchronous transmitter and receiver
- Half duplex, full duplex,, only TX and only RX modes
- Two wire – Transmit (TX) and Receive (RX)
- No Clock line
- Typically used baud rates – 9600 to 115200 bps
- Additional pins – flow control functionality
- Usually between two devices

I2C

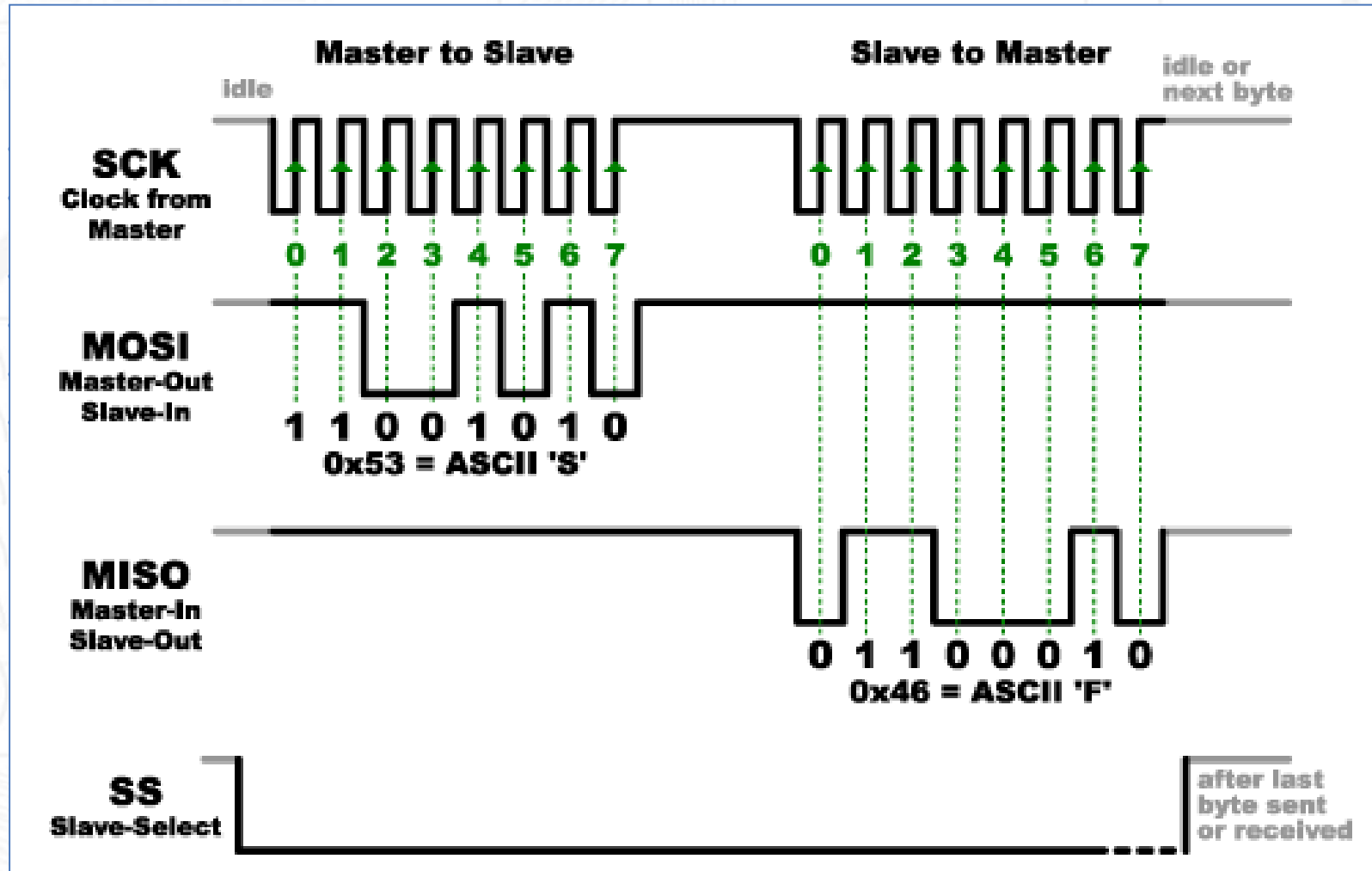
- Inter-integrated circuit (IIC / I²C)
- Half duplex protocol
- Two wire – Serial Data (SDA) and Serial Clock (SCL)
- Data rates from 100 kbps to 1000kbps
- A master can talk to 127 slaves

SPI

- Serial peripheral interface
- Full duplex protocol
- Four wire –
 - Master Out Slave In (MOSI)
 - Master In Slave Out (MISO)
 - Serial clock (SCK)
 - Slave Select (SS)
- Typically used data rates – 1 Mbps to 8 Mbps



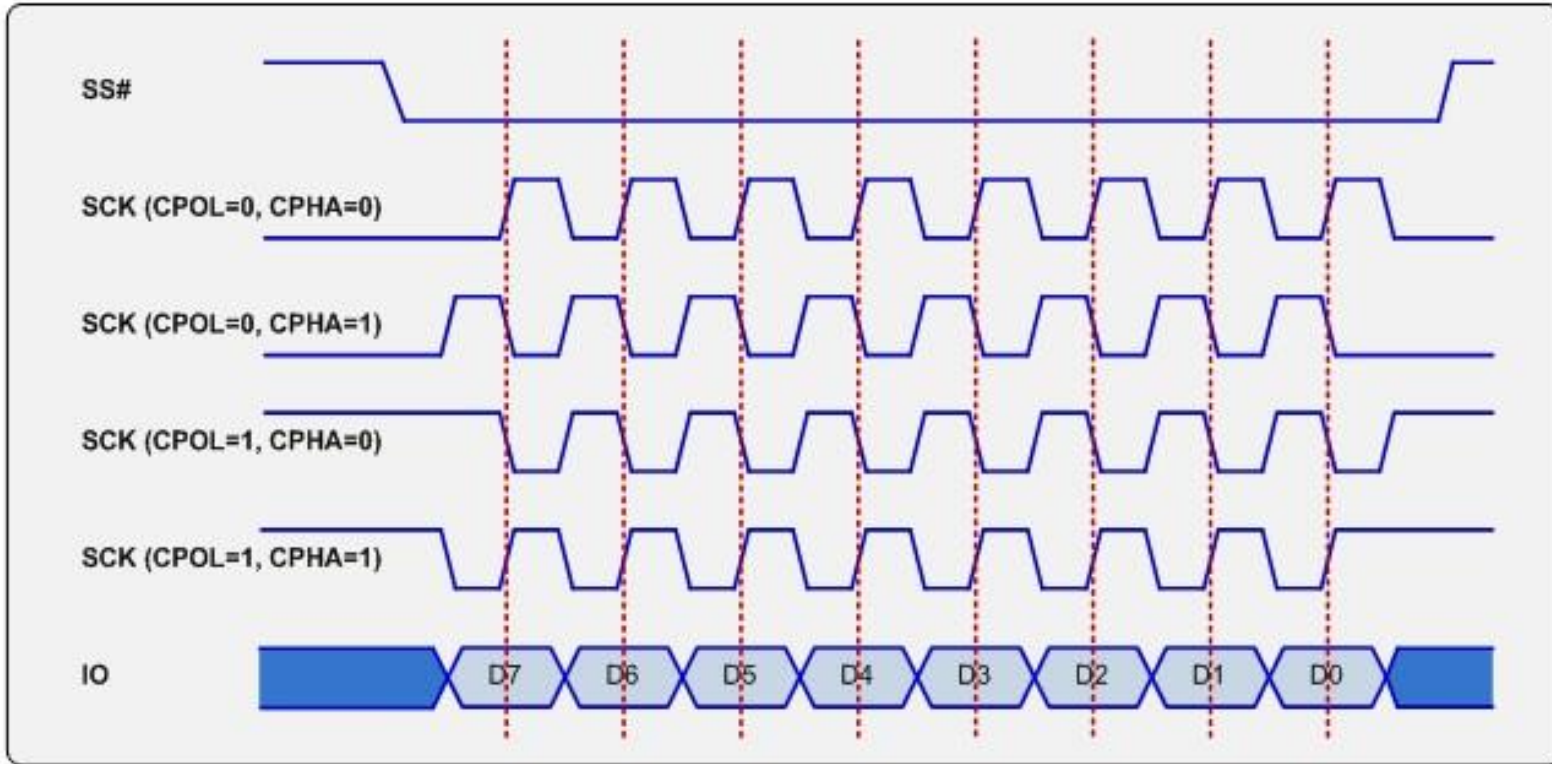
Serial Communication Block



SPI

- Serial peripheral interface
- Full duplex protocol
- Four wire –
 - Master Out Slave In (MOSI)
 - Master In Slave Out (MISO)
 - Serial clock (SCK)
 - Slave Select (SS)
- Typically used data rates – 1 Mbps to 8 Mbps

Serial Communication Block



SPI

- Serial peripheral interface
- Full duplex protocol
- Four wire –
 - Master Out Slave In (MOSI)
 - Master In Slave Out (MISO)
 - Serial clock (SCK)
 - Slave Select (SS)
- Typically used data rates – 1 Mbps to 8 Mbps

UART

Exercise 8:

- ◆ Use the project from Exercise 6 (Generate PWM signal of 100 Hz frequency with 10% duty cycle)
- ◆ Print the PWM parameters (duty cycle, compare value, period) on a PC terminal using UART
- ◆ Control PWM duty cycle from the PC (increase or decrease by 10% upon two different keypresses)

Solution key: Use `mtb_02_ex06_counter_dutycycle` project for reference.

Useful APIs:

Retarget-IO Middleware – `cy_retarget_io_init(CYBSP_DEBUG_UART_TX, CYBSP_DEBUG_UART_RX, 115200);`

Then make use of standard IO library functions like `printf`, `sprintf` etc. to read or print something to the terminal.

I2C

Exercise 9:

Control PWM brightness by writing data from an I2C master (use KitProg3 as master; using Bridge Control panel (BCP) on PC)

Solution key: Refer [mtb_02_ex07_i2c_brightness_control](#) project for this solution.

Useful APIs:

```
/* Allocate and initialize a I2C resource and auto select a clock */
cyhal_i2c_init(&i2c_slave, CYBSP_I2C_SDA, CYBSP_I2C_SCL, NULL);

/* Configure the I2C resource to be slave */
cyhal_i2c_configure (&i2c_slave, &i2c_slave_cfg);

/* Configure I2C slave write buffer for I2C master to write into */
cyhal_i2c_slave_config_read_buff(&i2c_slave, i2c_write_buffer, SL_WR_BUFFER_SIZE);

/* Configure I2C slave read buffer for I2C master to read from */
cyhal_i2c_slave_config_write_buff(&i2c_slave, i2c_read_buffer, SL_RD_BUFFER_SIZE);
```

SPI

Exercise 10:

Setup PSoC as both an SPI master and slave. SPI master sends a command every second to a SPI slave to toggle the LED. Use HAL APIs.

Solution key: Refer `mtb_02_ex08_spi_master` project for this solution.

Useful APIs:

`cyhal_spi_init()` – Initializes the SPI block and configures it as slave or master.

`cyhal_spi_set_frequency()` – set the SPI baud rate

`cyhal_spi_send()` – Sends the command

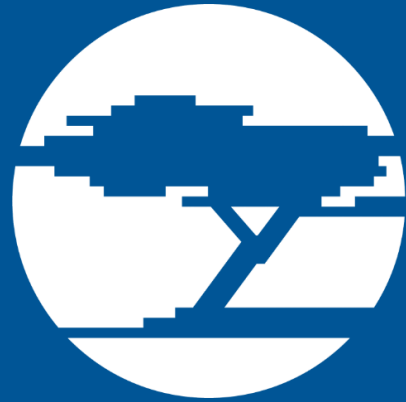
`cyhal_spi_recv()` – Receives the command

Resources

- [ModusToolbox User Guide](#)
- [Cypress Github Landing Page](#)
- [PSoC6 Architecture TRM](#)

Contact Information

- <https://community.cypress.com/welcome>
- Send your queries to ddka@cypress.com



CYPRESS[®]
EMBEDDED IN TOMORROW[™]